Mustererkennung - Aufgabenblatt 08

André Hacker und Dimitri Schachmann

1. Offline vs. Online

Den Code der Aufgabe 1 und 2 löst haben wir diesmal im Anhang am Ende. Wir haben zuerst die learnNeural Methode um Parameter itemsPerIteration erweitert, der den Algorithmus generalisiert. Übergibt man 1, hat man Online Learning, übergibt man N (Anzahl der Datensätze), hat man klassiches Offline/Batch Learning. Übergibt man eine Zahl L dazwischen, wird in jeder Iteration ein L-Subset der Daten zufällig gewählt und für dieses die Gradienten berechnet. So erreichen wir maximale Flexibilität.

Um Online und Offline Performance vergleichen zu können, betrachten wir bei Offline die effektiven Iterationen, also wie viele Gradienten berechnet wurden. Das ist nämlich die Zahl, die ausschlaggebend für die Laufzeit ist. Bei Offline-Learning mit 100 Iterationen und 7000 Testdaten haben wir 700.000 effektive Iterationen.

Außerdem haben wir eine logarithmische Skala für die Anzahl der Iterationen gewählt, um besser plotten zu können.

Die Messergebnisse ergeben sich aus völlig unabhängigen Versuchen, weshalb sich Schwankungen (hier bei Ionen-Daten) ergeben. Wir haben also erst 1 Iteration durchgeführt, das Ergebnis gemessen, dann mit neuen zufälligen Werten gestartet, mehr Iterationen durchgeführt und wieder das Ergebnis gewertet. Das hätte man auch anders machen können (nur einen Versuch, und zwischendrin die Successrate berechnen). Unser Ansatz hat den Vorteil, dass die großen Schwankungen bei den Versuchen (hängen mit initialen Gewichten und lokalen Minima zusammen) auch sichtbar werden.
Die Ergebnisse zeigen, dass Online-Learning effektiver ist, was auch die gängige Meinung der Literatur ist (vergleiche Bishop):

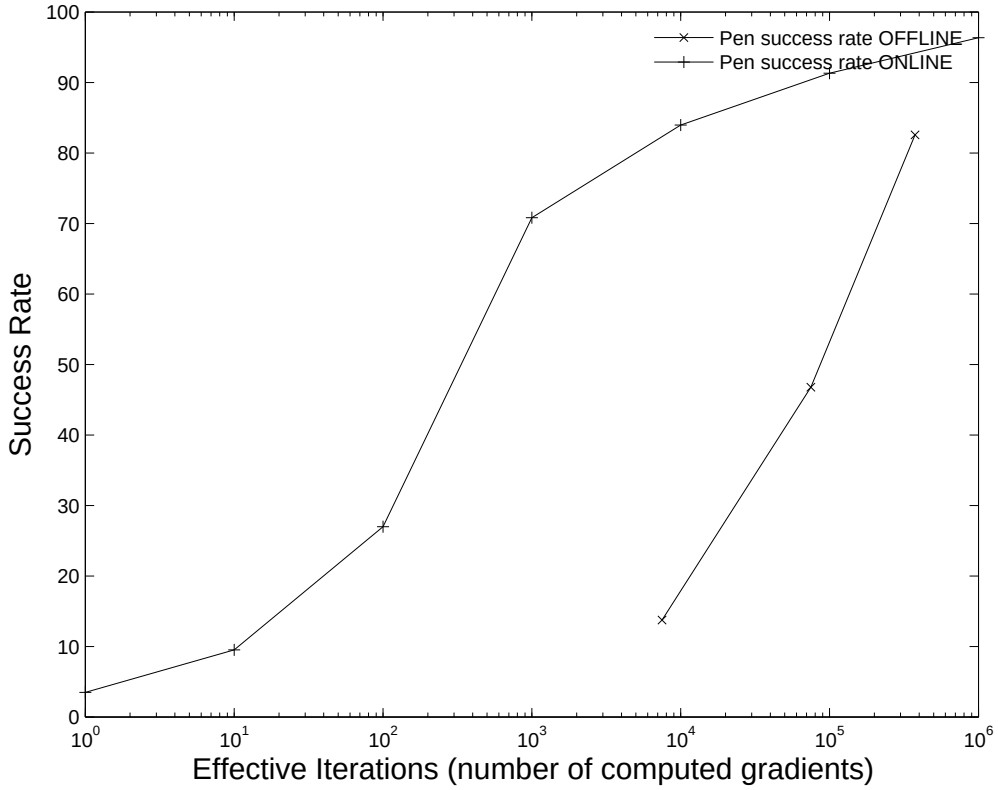

Abbildung 1: Pendigits Offline vs Online Performance

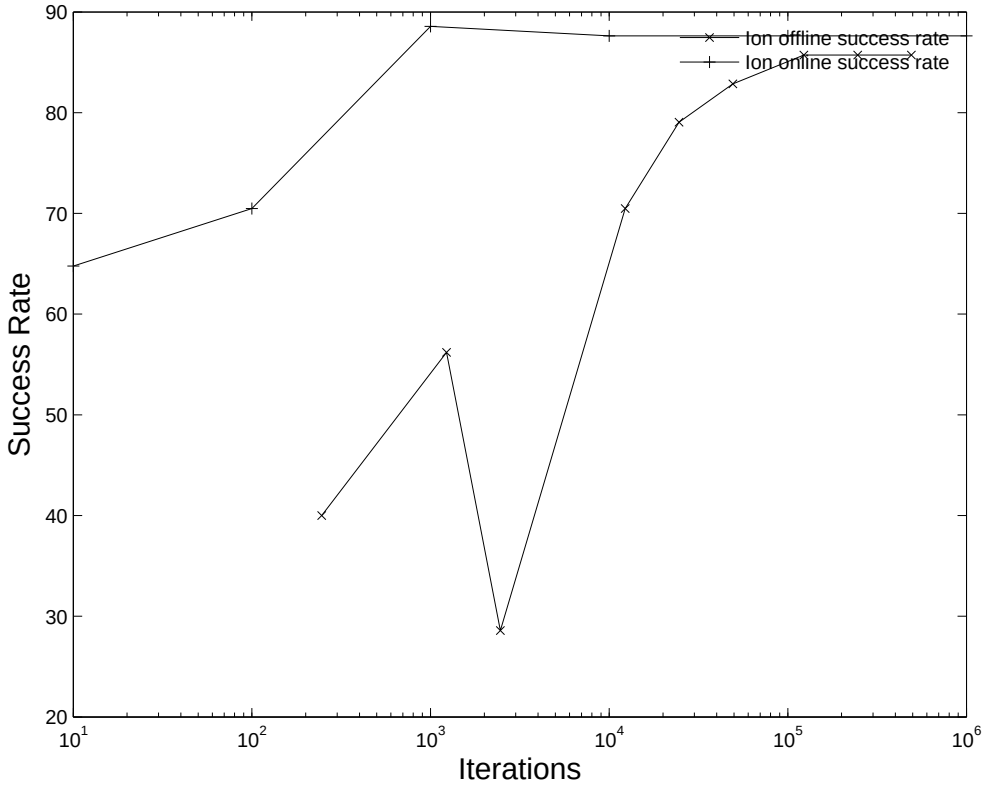

Abbildung 2: Ion Offline vs Online Performance

2. Rprop

Für die implementierung des Rprop Algorithmus haben wir schlicht den BP Code erweitert. Der learnNeural Funktion werden zusätzlich die parameter

- learnRateMax
- learnRateMin
- rprop_u
- rprop_d
- doRprop

übergeben. Wenn doRprop == 1, dann wird der Rprop Algorithmus durchgeführt, ansonsten der gewöhnliche BP. Es ist wichtig darauf zu achten, dass für Rprop batch learning aktiviert sein muss. Unsere Testläufe haben ergeben, dass unsere Rprop Implementierun bei bereits bei geringerer Iterationszahl besser ist, als BP bei der gleichen Iterationszahl. Es war jedoch schwierig gute Konstanten zu finden. Bei den pendigits daten war das schwieriger als bei den Ionendaten.
Wie erwartet führt der RPROP wesentlich schneller zu guten Werten.

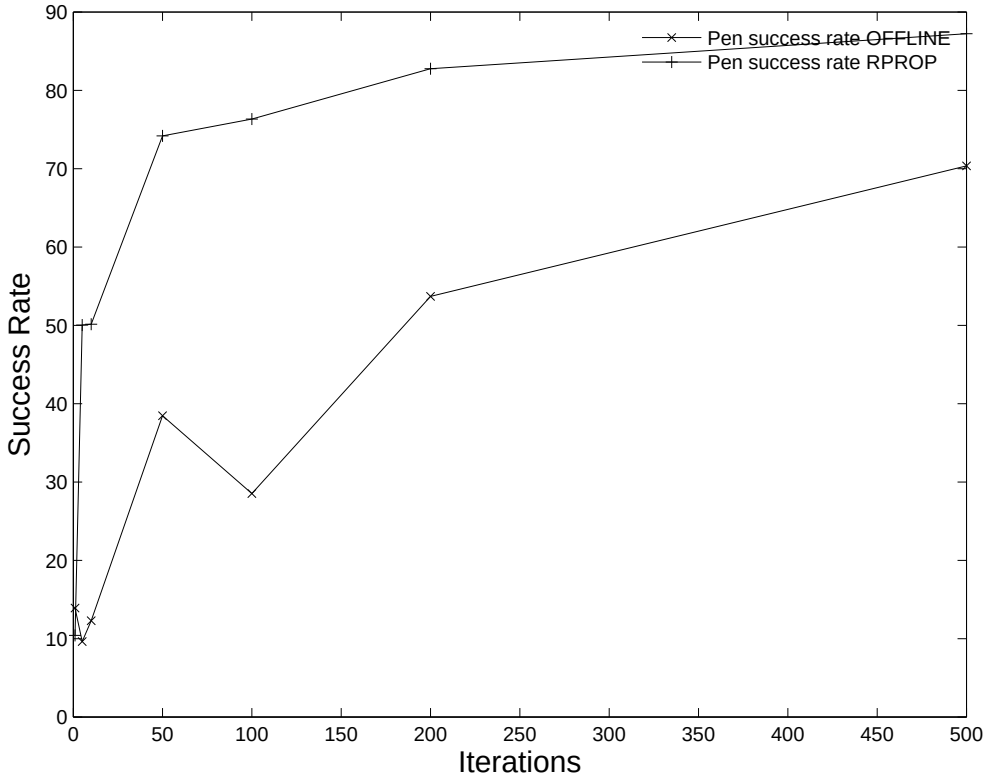

Abbildung 3: Pendigits Offline vs RPROP Performance

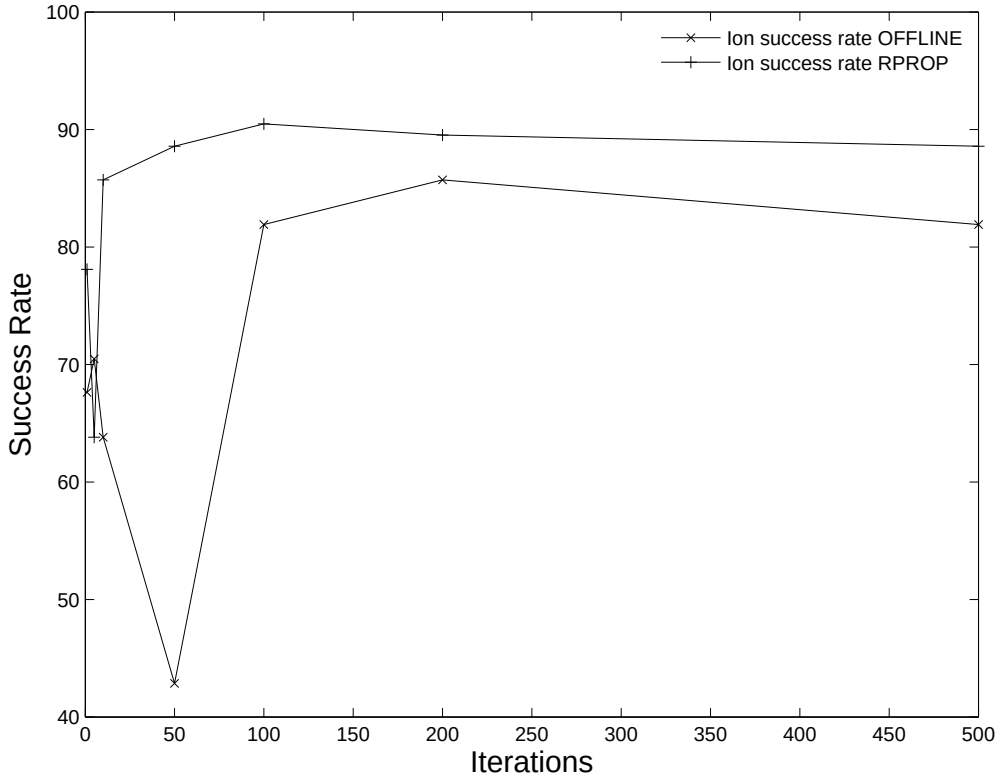

Abbildung 4: Ion Offline vs RPROP Performance

3. Implementierung

```
% TODO: Define error-threshhold for termination of learning
% TODO: Handle local minima: Start with multiple random weights. For each, do maxIteration iterations a

function bprop()

% Task 2 - Learn Pendigits
    task1();
end

function task1()

    % If weights are bad, more iterations don't help us (we often stuck in a local minimum)
    % => Try often with different random initial weights, take best (to be implemented)

    % -------
    % TASK 1 Compare Online and Offline
    % -------
    penTra = load('pendigits.tra');
    penTes = load('pendigits.tes');
    ion = load('ionosphere.data');  %34 features, 351 items, feature 2 always 0
    [ionTra ionTes] = randomPartition(ion, 0.7);
    penTraX = penTra(:,1:end-1);
    penTesX = penTes(:,1:end-1);
```

```
ionTraX = ionTra(:,1:end-1);
ionTesX = ionTes(:,1:end-1);

% Normalize to deviation from std-dev
penTraX = standardize(penTraX);
penTesX = standardize(penTesX);
ionTraX = standardize(ionTraX);
ionTesX = standardize(ionTesX);

% Convert output to dummy variable
penTraY = toDummy(penTra(:,end));
penTesY = toDummy(penTes(:,end));
ionTraY = toDummy(ionTra(:,end));
ionTesY = toDummy(ionTes(:,end));

% PCA (does not improve results)
%pcs = principalComponents(penTraX);
%penTraX = transformData(pcs, penTraX, 14);
%penTesX = transformData(pcs, penTesX, 14);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% TASK 1 ONLINE VS OFFLINE
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Global settings for task 1
doRprop = 0;
weightsFrom = -1;  % Interval for random weights
weightsTo = 1;
learnRateMax = 3;
learnRateMin = 0.1^3;
rprop_u = 1.5;
rprop_d = 0.8;
hiddenPen = 20;  % Hidden nodes
hiddenIon = 40;  % Hidden nodes

%%%%%%
% Plot ION Online vs Offline
%%%%%%
if 1==0
% ONLINE ION
learnRate = 0.1;  % Learning rate
itemsPerIteration = 1;
ionOnlineIts = 10.^[1:6];
sIonOnline = runTests(ionOnlineIts, ionTraX, ionTraY, ionTesX, ionTesY, hiddenIon, learnRate, items

% OFFLINE ION
learnRate = 0.001;  % Learning rate
itemsPerIteration = size(ionTraX,1);
ionOfflineIts = [1 5 10 50 100 200 500 1000 2000];
sIonOffline = runTests(ionOfflineIts, ionTraX, ionTraY, ionTesX, ionTesY, hiddenIon, learnRate, it

% Plot ION Online vs Offline
h = figure();
```

```
semilogx(ionOfflineIts.*size(ionTraX,1), sIonOffline', '-xr', 'Displayname', 'Ion success rate OFFL
hold on;
semilogx(ionOnlineIts, sIonOnline', '-+r', 'Displayname', 'Ion success rate ONLINE');
xlabel('Effective Iterations (number of computed gradients)', 'FontSize', 15);
ylabel('Success Rate', 'FontSize', 15);
legend('show');
legend('boxoff');
print(h,'-deps','task1-ion.eps');

return

%%%%%%
% Plot PEN Online vs Offline
%%%%%%

% ONLINE PEN
itemsPerIteration = 1;
learnRate = 0.1;
penOnlineIts = 10.^[0:5];
sPenOnline = runTests(penOnlineIts, penTraX, penTraY, penTesX, penTesY, ...
      hiddenPen, learnRate, itemsPerIteration, weightsFrom, weightsTo,learnRateMax, ...
      learnRateMin, rprop_u, rprop_d, doRprop)

% OFFLINE PEN
% best Learning rate for batch (pen): 0.001
itemsPerIteration = size(penTraX,1);
learnRate = 0.001;  % best Learning rate for batch (pen): 0.001
penOfflineIts = [1 10];
sPenOffline = runTests(penOfflineIts, penTraX, penTraY, penTesX, penTesY, ...
        hiddenPen, learnRate, itemsPerIteration, weightsFrom, weightsTo,learnRateMax, ...
        learnRateMin, rprop_u, rprop_d, doRprop)

% PLOT
h = figure();
semilogx(penOfflineIts.*size(penTraX,1), sPenOffline', '-xr', 'Displayname', 'Pen success rate OFFL
hold on;
semilogx(penOnlineIts, sPenOnline', '-+r', 'Displayname', 'Pen success rate ONLINE');
xlabel('Effective Iterations (number of computed gradients)', 'FontSize', 15);
ylabel('Success Rate', 'FontSize', 15);
legend('show');
legend('boxoff');
print(h,'-deps','task1-pen.eps');

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% TASK 2 RPROP VS OFFLINE
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Global settings for this task
itemsPerIteration = size(ionTraX,1);
learnRateMax = 3;
learnRateMin = 0.001;
```

```
rprop_u = 1.5;
rprop_d = 0.8;

%%%%%%
% Plot PEN Online vs Offline
%%%%%%

% RPROP PEN
doRprop = 1;
learnRate = 0.1;
penRpropIts = [1 5 10 50 100 200 500];
sPenRprop = runTests(penRpropIts, penTraX, penTraY, penTesX, penTesY, ...
      hiddenPen, learnRate, itemsPerIteration, weightsFrom, weightsTo,learnRateMax, ...
      learnRateMin, rprop_u, rprop_d, doRprop)

% OFFLINE PEN
% best Learning rate for batch (pen): 0.001
doRprop = 0;
learnRate = 0.001;  % best Learning rate for batch (pen): 0.001
penOfflineIts = penRpropIts;
sPenOffline = runTests(penOfflineIts, penTraX, penTraY, penTesX, penTesY, ...
        hiddenPen, learnRate, itemsPerIteration, weightsFrom, weightsTo,learnRateMax, ...
        learnRateMin, rprop_u, rprop_d, doRprop)

% PLOT
h = figure();
plot(penOfflineIts, sPenOffline', '-xr', 'Displayname', 'Pen success rate OFFLINE');
hold on;
plot(penRpropIts, sPenRprop', '-+r', 'Displayname', 'Pen success rate RPROP');
xlabel('Iterations', 'FontSize', 15);
ylabel('Success Rate', 'FontSize', 15);
legend('show');
legend('boxoff');
print(h,'-deps','task2-pen.eps');
return

%%%%%%
% Plot ION RPROP vs Offline
%%%%%%

% RPROP ION
doRprop = 1;
learnRate = 0.01;  % Learning rate
ionRpropIts = [1 5 10 50 100 1000];
sIonRprop = runTests(ionRpropIts, ionTraX, ionTraY, ionTesX, ionTesY, hiddenIon, learnRate, itemsPe

% OFFLINE ION
doRprop = 0;
learnRate = 0.001;  % Learning rate
ionOfflineIts = ionRpropIts;
sIonOffline = runTests(ionOfflineIts, ionTraX, ionTraY, ionTesX, ionTesY, hiddenIon, learnRate, ite

% Plot ION Online vs Offline
```

```
    h = figure();
    plot(ionOfflineIts, sIonOffline', '-xr', 'Displayname', 'Ion success rate OFFLINE');
    hold on;
    plot(ionRpropIts, sIonRprop', '-+r', 'Displayname', 'Ion success rate RPROP');
    xlabel('Iterations', 'FontSize', 15);
    ylabel('Success Rate', 'FontSize', 15);
    legend('show');
    legend('boxoff');
    print(h,'-deps','task2-ion.eps');

end




% A row vector with all number of iterations to execute
function s = runTests(its, TraX, TraY, TesX, TesY, ...
                      hidden, learnRate, itemsPerIteration, weightsFrom, weightsTo, ...
                      learnRateMax, learnRateMin, rprop_u, rprop_d, doRprop)
    fid = fopen('task1-results.txt','a');
    fprintf(fid, '\nStart Testrun\n');
    s=[];
    for i=1:size(its,2)
        maxIterations = its(i);
        errEvery = maxIterations;

        tic()
        [W1d W2d Eabs Esq Eonline] = learnNeural(TraX, TraY, hidden, size(TraY,2), ...
                                                 learnRate, maxIterations, errEvery, ...
                                                 itemsPerIteration, weightsFrom, weightsTo, ...
                                                 learnRateMax, learnRateMin, rprop_u, rprop_d, doRprop)
        elapsed = mytoc('Time for learning (seconds): ');
        s = [s; neuralTestSuccess(W1d, W2d, TesX, TesY)];

        fprintf(fid, 'Pen Its: %.0f\tItemsPerIt: %.0f\tLearnRate: %.4f\tHidden: %.0f\tSeconds: %.2f\n'
                itemsPerIteration, learnRate, hidden, elapsed);
        fprintf(fid, '> Success Rate: %.2f\n', s(i,1));
    end
    fclose(fid);
end

% Splits data into training and test set
% traPercentage: From 0 to 1
function [tra tes] = randomPartition(data, traPercentage)
    N = size(data,1);
    traIds = randsample(N, round(N*traPercentage));
    tra = data(traIds, :);
    tesIds = setdiff([1:N],traIds);
    tes = data(tesIds, :);
end

% Classify input and compute success rate
% W1, W2 = weights with bias
```

```
% X = input, without bias
% Y = rows of dummy variables with result
% sr = success rate in percentages
function sr = neuralTestSuccess(W1d, W2d, X, Y)
    Xd = [X, ones(size(X,1), 1)];
    isVsShould = zeros(size(X,1),2);
    for i = 1:size(X,1)
        [value realClass] = max(Y(i,:));
        realClass = realClass - 1;
        predictedClass = classifyNeural(W1d, W2d, Xd(i,:));
        isVsShould(i,:) = [predictedClass realClass];
    end
    diff = isVsShould(:,1) - isVsShould(:,2);
    right = size(diff(diff==0));
    sr = right / size(diff) * 100;
end

% Returns a 1xM vector
function r = evalNetwork(W1d, W2d, xd)
    o1 = sigmoid(xd*W1d);   % 1xK
    o1d = [o1 1];
    r = sigmoid(o1d*W2d);  % 1xM
end

function c = classifyNeural(W1d, W2d, xd)
    dummyOut = evalNetwork(W1d, W2d, xd);
    [value index] = max(dummyOut);
    c = index-1;
end

% Input: Nx1 vector with scalar values (classes)
%   Assumption: Classes are integers (Class 1 = value 1)
% Output: NxC vector with dummy variable in each row
function R = toDummy(Y)
    numberClasses = max(Y);
    R = zeros(size(Y,1), numberClasses);
    for i=1:size(Y,1)
        R(i, Y(i)+1) = 1;
    end
end

% Learns the weights of a 2 Layer Neural Network
% Uses Online (sequential) Training
% @param X: Samples
% @param Y: Output (encoded as dummy variables)
% @param K: Number of hidden nodes
% @param M: Number of output nodes
% @param learnRate: Learning rate
% @param maxIterations: when to stop
% @param errEvery: Error (for all items) will be calculated only every ... iterations.
%                  Error computation is very costly, so we don't want to do it in every step
% @param itemsPerIteration: Number of gradients to compute in each iteration (1=Online, N=Offline).
%                  Allows semi-batch (value between 1 and N)
```

```
% @param learnRateMax, learnRateMin, rprop_u, rprop_d, doRprop: Parameters for RPROP.
%                  To enable, set doRprop=1
% @return W1d: weights of the 1st layer
% @return W2d: weights of the 2nd layer
% @return Eabs: Matrix. Eabs(i,j) is the absolute avarage deviation
%   (abs(predicted - y)) for the j-th output unit in iteration i. For
%   all samples
% @return Esq: Matrix. Esq(i,j) is the half squared avarage deviation
%   (((predicted - y)^2)/2) for the j-th output unit in iteration i. For
%   all samples
% @return Eonline: column vector. Eonline(i) is the the half
%   squared error for the current random sample in iteration i of
%   online learning. That is the function that we optimize for.
function [W1d W2d Eabs Esq Eonline] = learnNeural(X, Y, ...
                                                  K, M, learnRate, maxIterations, errEvery, ...
                                                  itemsPerIteration, weightsFrom, weightsTo, ...
                                                  learnRateMax, learnRateMin, rprop_u, rprop_d, doRprop

    D = size(X, 2);   % number of input nodes (dimensionality of input)
    N = size(X, 1);   % number of input items
    Xd = [X, ones(size(X, 1), 1)];  % convenience

    Eabs = zeros(ceil(maxIterations/errEvery), M); % Track global (for all estraining items) absolute d
    Esq = zeros(ceil(maxIterations/errEvery), M);
    Eonline = zeros(maxIterations, 1);

    % Start with random initial weights
    % 1st col = all weights to 1st node
    W1 = rand(D, K)*(weightsTo-weightsFrom) + weightsFrom;
    W2 = rand(K, M)*(weightsTo-weightsFrom) + weightsFrom;
    W1d = [W1; rand(1, K)*(weightsTo-weightsFrom) + weightsFrom];
    W2d = [W2; rand(1, M)*(weightsTo-weightsFrom) + weightsFrom];


    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    LearningRatesW1 = ones(D+1, K);
    LearningRatesW2 = ones(K+1, M);
    partialDerivativesW1_old = zeros(D+1, K);
    partialDerivativesW2_old = zeros(K+1, M);

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    for i=1:maxIterations
        % Error-Tracking (optional)
        if (mod(i-1, errEvery) == 0)
            tmp=zeros(N, M);
            for l=1:N
                tmp(l,:) = evalNetwork(W1d, W2d, Xd(l,:));
            end
            Eabs((i-1)/errEvery+1, :) = sum(abs(tmp - Y), 1);
            Esq((i-1)/errEvery+1, :) = sum(((tmp - Y).^2),1) / 2;
        end
```

```
% 0) Choose inputs for this iteration
if itemsPerIteration > 1
    if itemsPerIteration >= N
        % Offline Learning
        XSub = X;
        XdSub = X;
        YSub = Y;
    else
        % Offline Learning, but only for subset
        % We could always do this, but it's slows down other cases
        ids = randsample(N,itemsPerIteration);
        XSub = X(ids, :);
        XdSub = Xd(ids, :);
        YSub = Y(ids, :);
    end
else
    % 0) Online learning: Pick one random input
    id = randi(size(X,1));
    XSub = X(id, :);
    YSub = Y(id, :);
end


partialDerivativesW1 = zeros(D+1, K);
partialDerivativesW2 = zeros(K+1, M);
DeltaW1d = zeros(D+1, K);
DeltaW2d = zeros(K+1, M);
for j=1:itemsPerIteration % For offline learning

    x = XSub(j, :);
    xd = [x 1];
    y = YSub(j, :);

    % 1) Forward Propagation:
    % 1.1) Compute Layer hidden layer output o1 and Output layer output o2
    o1 = sigmoid(xd*W1d);   % 1xK
    o1d = [o1 1];
    o2 = sigmoid(o1d*W2d);  % 1xM
    o2d = [o2 1];

    % 1.2) Compute Derivations for Hidden units (D1) and Output Units (D2) (diagonal matrix)
    D1 = diag(o1.*(1-o1));   % KxK
    D2 = diag(o2.*(1-o2));   % MxM

    % 1.3) Compute Derivation DE for Error unit
    DE = (o2-y)';  % Mx1
                   %Eonline(i) = sum((DE.^2)/2)/M;  % TODO: Do this only once for Online!

    % 2) Back Propagation
     % to Output Layer
    backErr2 = D2*DE;
    % to Hidden Layerm
```

```
            backErr1 = D1 * W2 * backErr2;

            partialDerivativesW2 = partialDerivativesW2 + (backErr2 * o1d)';
            partialDerivativesW1 = partialDerivativesW1 + (backErr1 * xd)';

        end
        if(doRprop)
            %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
            % output layer
            % g is negative if the gradient direction changed
            g = partialDerivativesW1.*partialDerivativesW1_old;
            % utility
            lmax = repmat(learnRateMax,size(LearningRatesW1));
            lmin = repmat(learnRateMin,size(LearningRatesW1));
            ru = repmat(rprop_u,size(LearningRatesW1));
            rd = repmat(rprop_d,size(LearningRatesW1));

            % compute the new learning rates according to formula
            LearningRatesW1 = min(LearningRatesW1.*ru, lmax).*(g>0) +  max(LearningRatesW1.*rd, lmin).*
            % save the gradient if it had the same direction
            partialDerivativesW1_old = partialDerivativesW1.*(g>=0);
            % compute the weight changes
            DeltaW1d = (((-sign(partialDerivativesW1)).*LearningRatesW1).*(g>=0));

            %%% hidden layer
            g = partialDerivativesW2.*partialDerivativesW2_old;
            lmax = repmat(learnRateMax,size(LearningRatesW2));
            lmin = repmat(learnRateMin,size(LearningRatesW2));
            ru = repmat(rprop_u,size(LearningRatesW2));
            rd = repmat(rprop_d,size(LearningRatesW2));

            LearningRatesW2 = min(LearningRatesW2.*ru, lmax).*(g>0) +  max(LearningRatesW2.*rd, lmin).*
            partialDerivativesW2_old = partialDerivativesW2.*(g>=0);
            % compute the weight changes
            DeltaW2d = (((-sign(partialDerivativesW2)).*LearningRatesW2).*(g>=0));
            %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        else
            % 3) Compute weight adoptions
            DeltaW2d = (-learnRate' * partialDerivativesW2);
            DeltaW1d = (-learnRate' * partialDerivativesW1);
        end
        % 4) Update adoptions
        W2d = W2d + DeltaW2d;
        W1d = W1d + DeltaW1d;
        W2 = W2d(1:end-1,:);
        W1 = W1d(1:end-1,:);
    end
end

% Logistic sigmoid function
function r = sigmoid(x)
    r = 1./(1 + exp(-x));
end
```

```
% Standardize data by subtracting the mean and deviding by the standard
% deviation
function r = standardize(samples)
    m = mean(samples);
    sd = std(samples);
    % Attention: sd may never be 0!
    sd(sd==0)=0.0000001;
    for i = 1:size(samples,1)
        samples(i,:) = (samples(i,:) - m) ./ sd;
    end
    r = samples;
end

% trasform data to a new basis, given by eigenspace (eigenvectors)
function r = transformData(eigenspace, data, dim)
    r = (eigenspace(:,end-dim+1:end)'*data')';
end

% computes the principal components for the given data
% r = eigenvectors of the covariance matrix
function r = principalComponents(data)
    covarMatrix = cov(data);
    [r eigen_values] = eig(covarMatrix);
end

% Plot Error rate over iterations
function plotErrorForIterations(E, name, xtitle, ytitle)
    h = figure();
    hold on;
    xlabel(xtitle, 'FontSize', 15);
    ylabel(ytitle, 'FontSize', 15);
    for i=2:(size(E,2))
        plot(E(:,1),E(:,i));
    end
    print(h,'-deps',[name '.eps']);
end


function elapsed = mytoc(timerDescription)
    global total;
    %disp( ceil(toc()) )
    elapsed = toc();
    fprintf( '%s %.6f\n', timerDescription, elapsed );
    %printf([timerDescription mat2str(ceil(toc()))]);
end
```